• Remember to register your attendance using the UoB Check-In app. Either

1. download, install, and use the native app[a] available for Android and iOS, or
2. directly use the web-based app available at

https://check-in.bristol.ac.uk

noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.

• The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

https://www.bristol.ac.uk/it-support

• The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

https://cs-uob.github.io/COMS10015/vm

• The questions are roughly classified as either C (for core questions, that *should* be attempted within the lab. slot), A (for additional questions, that *could* be attempted within the lab. slot), or R (for revision questions). Keep in mind that we only *expect* you to attempt the C-class questions: the other classes are provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring them.

• There is an associated set of solutions is available, at least for the C-class questions. These solutions are there for you to learn from (e.g., to provide an explanation or hint, or illustrate a range of different solutions and/or trade-offs), rather than (purely) to judge your solution against; they often present *a* solution vs. *the* solution, meaning there might be many valid approaches to and solutions for a question.

• Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance

[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.

[c] https://www.vagrantup.com

COMS10015 lab. worksheet #4

Before you start work, download (and, if need be, unarchive[a]) the file

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/lab-04_q.tar.gz

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a]For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-04_q.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive`→`Open` menu item to open `lab-04_q.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-04_q.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b]For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

§1. C-class, or core questions

▷ **Q1[C].** In the lecture slot(s), we studied the design of a cyclic n-bit counter, whose behaviour is such that the output r steps through values $0, 1, \ldots, 2^n - 1, 0, 1, \ldots$, i.e., it counts from 0 up to $2^n - 1$ (the maximum value which we can represent in n bits) then cycles back to 0 again. Assuming $n = 4$, use LogisimEvo to implement and simulate variants of this design based on

a latches, *and*

b flip-flops.

Keep in mind that a central goal of doing so is improved understanding. Versus purely combinatorial logic, understanding sequential logic design is difficult in the sense it depends on changes to the state over time; interacting with it in practice can offer a *much* more effective explanation than in theory therefore. As a result, the (strong) recommendation is to make use of built-in components provided by LogisimEvo. Although it provides *most* components required, the archive includes an additional user-defined a 2-phase clock generator component: this can be used to generate Φ_1 and Φ_2, i.e., the non-overlapping clock signals, required for the latch-based variant.

▷ **Q2[C].** Consider an alternative to the cyclic n-bit counter design, which instead exhibits saturating[1] behaviour. The idea is to update the adder so it produces

$$r = \begin{cases} x + y & \text{if } x + y < 2^n \\ 2^n - 1 & \text{if } x + y \geq 2^n \end{cases}$$

Such an update means that if $x + y$ *can* be represented in n bits then the adder produces $x + y$, but if $x + y$ *cannot* be represented in n bits then the adder produces the maximum value that can; rather than the counter output step through values $0, 1, \ldots, 2^n - 1, 0, 1, \ldots$, therefore, it instead steps through values $0, 1, \ldots, 2^n - 1, 2^n - 1, 2^n - 1, \ldots$, because the adder output eventually becomes "clamped" to $2^n - 1$. Assuming $n = 4$, use LogisimEvo to implement and simulate a variant of this design based on

a latches, *or*

b flip-flops.

§2. R-class, or revision questions

▷ **Q3[R].** There is a set of questions available at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/misc-revision_q.pdf

Using pencil-and-paper, each asks you to solve a problem relating to Boolean algebra. There are too many for the lab. session(s) alone, but, in the longer term, the idea is simple: attempt to answer the questions, applying theory covered in the lecture(s) to do so, as a means of revising and thereby *ensuring* you understand the material.

[1]See, e.g., https://en.wikipedia.org/wiki/Saturation_arithmetic.

§3. A-class, or additional questions

▷ **Q4[A].** In the lecture slot(s) we saw that an SR-type latch design could be improved, step-by-step, to produce a D-type latch whose excitation table is

	Current		Next	
D	Q	$\neg Q$	Q'	$\neg Q'$
0	**?**	**?**	0	1
1	**?**	**?**	1	0

Doing so is useful, because the result 1) includes an enable signal, and 2) disallows the case where $S = R = 1$. Note that, as is, the design(s) presented used a *mixture* of logic gates (e.g., NOR, AND, and NOT): use LogisimEvo to implement and simulate a D-type latch variants based on

a NOR *only*, and

b NAND *only*,

having first developed an on-paper design for each.

References

[1] N.H.E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. 2nd ed. Addison Wesley, 1993 (see p. 4).

A Frequently Asked Questions (FAQs)

I don't understand the 2-phase clock generator design: can you explain? There are various ways to implement a 2-phase clock generator: it is not crucial to understand the specific design whose implementation has been provided, but *iff.* you are interested then Weste and Eshraghian [1, Figure 5.73] offer an explaination.

One common question is about why the NOT gates exist, i.e., given that we know $\neg\neg x \equiv x$, why are these gates included in the implementation? This touches on a more general difference between simulated and physical implementation. In short, within a simulated implementation there is indeed no need for the gates. But the reason is that the simulation does not consider the impact of gate delay which *will* be evident within an alternative, physical implementation. There, the gates allows a (controlled) delay to be introduced: given the goal, this is useful since it allows one to parameterise features of Φ_1 and Φ_2.

I'm confused by the recommended use of built-in components: what do you mean? First, *why*. The recommendation is simply to limit dependencies between lab. worksheets, plus the volume of work required: it is possible to rely on user-defined components instead, but doing so will probably complicates your solution in the sense you will naturally focus less on the central goal(s). Second, *which* and *how*:

- The built-in `Arithmetic`→`Adder` component can be useful as a way to compute the sum of two n-bit operands. Note that the value of n is controlled by the data bits property: this means it can be used to instantiate an n-bit ripple-carry adder for any n *or* a 1-bit full-adder (n instances of which could be used for form said n-bit ripple-carry adder).
- The built-in `Memory`→`D Flip-flop` component can be useful as a way to store 1-bit values. However, it is important to take care re. at least 2 points. First, note that this component is based on flip-flops, and so will be edge-triggered by default; changing the trigger property allows it to alternatively support, e.g., level-triggered latch. Second, this component has inputs and outputs which need explaination:
  - on the left-hand edge, there is a clock input: this represents what we referred to as enable or en.
  - on the top and bottom edge respectively, there is a preset and clear input:
- The built-in `Input/Output`→`Button` component can be useful as a way to model, e.g., reset or rst: it acts in a similar way to an input pin, but is automatically "unpressed" after being "pressed" using the poke tool.

I'm stuck with or confused about simulation for sequential logic designs?! Many of the central concepts are explained in detail by LogisimEvo documentation under the the `Help`→`User Guide` menu item: see the `Guide to Being a Logisim User`→`Menu reference`→`The Simulate menu` entry, for example. At a high level, the actual process of simulation can be described as two steps. First, use the `Simulate`→`Reset Simulation` menu item to reset the simulation state: doing so is important, because it ensures a fixed starting state for the simulation. Second, use one of two mechanism to advance the simulation:

1. The `Simulate`→`Tick Once` menu item manually provokes a clock transition, from 0 to 1 or from 1 to 0, moving simulation ahead in time by one half-cycle.
2. The `Simulate`→`Ticks Enabled` menu item enables/disables automatic clock transitions (which is analogous to automatically selecting `Simulate`→`Tick Once` again and again, in an infinite loop); once enabled, `Simulate`→`Ticks Frequency` can be used to control the frequency of transitions.

The task of debugging an implementation that contains sequential logic components, and whose state changes (semi-)automatically as the result of the features in a clock, can be tough. As with debugging software, a robust debugging strategy is vital: to understanding why some output or behaviour is incorrect, for example, is to vital to understand the state which led to it. Therefore, manual control of clock transitions is useful because it it allows easier inspection of intermediate state and hence controlled verification that the design does what it should at each step. The LogisimEvo probe component can also be useful: it allows the inspection of a wire value without adding a "dummy" output. The component will adapt to whatever wire type it is connected to, and allows output in a range of representations (e.g., binary, signed or unsigned decimal, or hexadecimal). Alternatively, you may find the logging functionality under the `Simulate`→`Logging ...` menu item useful.